Multithreading

Часто встречаются ситуации, в которых предпочтительно иметь несколько потоков (исполняющихся
параллельно или квазипараллельно) в одном адресном пространстве, как если бы они были различными отдельными процессами (однако разделяющими одно и то же адресное пространство).
В простейшем случае каждому процессу соответствует адресное пространство и одиночный управляющий поток

1. Основной причиной использования множества потоков является выполнение большинством приложений существенного числа действий.
2. Легкость создания и уничтожения потоков. На создание потока уходит примерно в 100 раз меньше времени, чем на создание процесса.
3. Третьим аргументом в пользу многопоточности является производительность.

Процесс можно рассматривать как способ объединения родственных ресурсов в одну группу.
У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы.
Ресурсами являются:

- открытые файлы;
- дочерние процессы;
- необработанные аварийные сообщения;
- обработчики сигналов, и м. д.

Гораздо проще управлять ресурсами, объединив их в форме процесса.
Процесс можно рассматривать как поток исполняемых команд или просто поток.
Компоненты потока:

- счетчик команд, отслеживающий порядок выполнения действий;
- регистры, в которых хранятся текущие переменные;
- стек, содержащий протокол выполнения процесса.

Отличия потока и процесса
Процессы используются для группирования ресурсов, а потоки являются объектами, поочередно исполняющимися на центральном процессоре.

Различные потоки в одном процессе не так независимы, как различные процессы. У всех потоков одно и то же адресное пространство, что означает совместное использование глобальных переменных. Но, конечно, каждый поток обладает собственным стеком.

Любой поток имеет доступ к любому адресу ячейки памяти в адресном пространстве процесса, один поток может считывать, записывать или даже стирать информацию из стека другого потока.

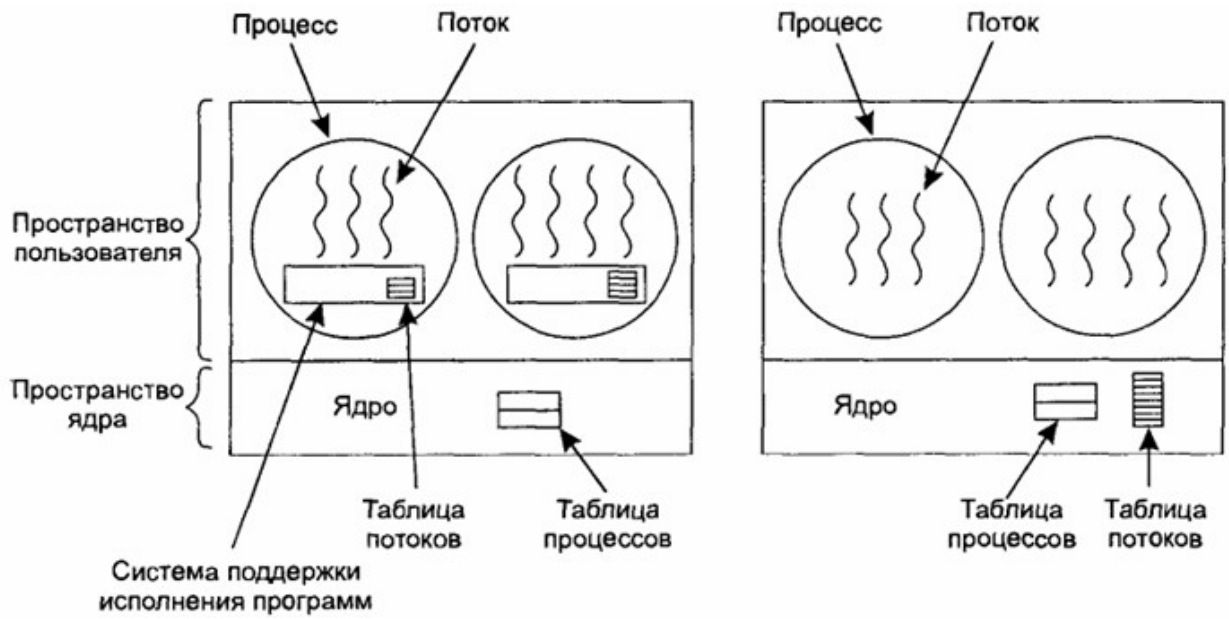
Элементы процесса	Элементы потока
Адресное пространство	**Счетчик команд**
Глобальные переменные	**Регистры**
Открытые файлы	**Стек**
Дочерние процессы	**Состояние**
Необработанные аварийные сигналы	
Сигналы и их обработчики	
Информация об использовании ресурсов	

Защиты не существует, поскольку: - это невозможно и - это ненужно. В отличие от различных процессов, которые инициированы различными пользователями, один процесс всегда запущен одним пользователем, и потоки созданы, чтобы работать совместно.

- **IEEE standard 1003.1c** — стандарт создания переносимых многопоточных программ.
- **Пакет Pthreads**, реализует работу с потоками, поддерживается большинством UNIX-систем.
- В стандарте определено более 60 вызовов функций.
- Все потоки Pthreads имеют определенные свойства.
  - У каждого потока есть свой идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре. Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.

Вызовы, связанные с потоком	Описание
pthread_create	Создание нового потока
pthread_exit	Завершение работы вызвавшего потока
pthread_join	Ожидание выхода из указанного потока
pthread_yield	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока

Есть два основных способа реализации пакета потоков: в **пространстве пользователя** и **ядре**.

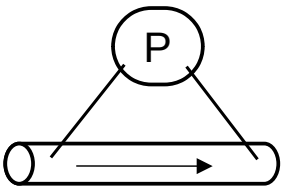


Pipes

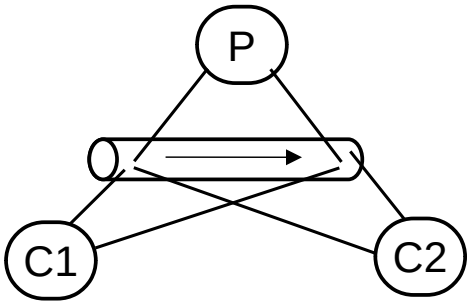
Конвейеры (в оригинале *pipes*) представляют собой традиционный и достаточно распространенный механизм взаимодействия процессов в Linux. Например, простейшая конструкция вида *who | sort* , запускаемая в *shell*, создает два конкурентных процесса, соединенных конвейером так, что поток вывода первого из них попадает в поток ввода второго.

Для организации однонаправленных конвейеров в приложениях применяется системный вызов *pipe()*.
В качестве параметра вызова выступает указатель на массив двух файловых дескрипторов.
Первый дескриптор связан с выходом образуемого конвейера, а второй - со входом.

В программе *whosortpipe* реализуется типовой подход к созданию однонаправленного канала передачи данных между процессами потомками.
Вначале родительский процесс *P* вызовом *pipe()* создает конвейер.

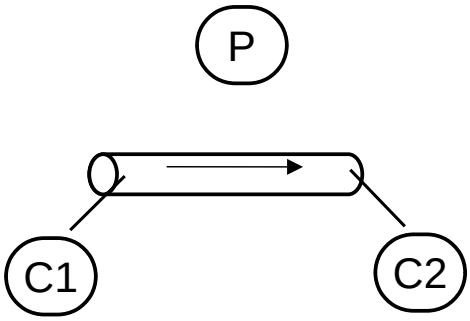


Затем порождает пару дочерних процессов, каждый из которых наследует от родителя оба открытых вызовом *pipe()* дескриптора.



Потомки *C1* и *C2* закрывают дескрипторы, соответствующие тем концам конвейера. В которых они не нуждаются, а на оставшиеся перенастраивают (ассоциируют) свой *стандартный вывод* и *стандартный ввод*, соответственно.

После того, как процесс родитель *P* со своей стороны, закроет оба дескриптора (отключится), в системе образуется однонаправленный канал, по которому в поток ввода одного дочернего процесса *C2* будут поступать данные из выходного потока другого процесса *C1*.



ПРИМЕЧАНИЕ:
Каждая запущенная из командного интерпретатора *shell* программа (команда) получает три открытых потока ввода/вывода:
- стандартный ввод (дескриптор 0),
- стандартный вывод (дескриптор 1),
- стандартный вывод ошибок (дескриптор 2).

По умолчанию все эти потоки ассоциированы с терминалом.
То есть любая программа, не использующая потоки, кроме стандартных, будет ожидать ввода с клавиатуры терминала, а весь вывод этой программы, включая сообщения об ошибках, будет происходить на экран терминала.

```
/* Программа whosortpipe.cpp */

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
main()
{
int     fds[2];
pipe(fds);   /* Создание конвейера */
        /* Один дочерний процесс подключает стандартный ввод stdin к
выходу конвейера
        и закрывает другой конец */
    if (fork()==0){
        dup2(fds[0], 0);   /* перенаправление ввода */
        close(fds[1]);
    execlp("sort", "sort", 0);
    }
/* Другой дочерний процесс подключает стандартный вывод stdout ко
входу конвейера
        и закрывает другой конец  */
    else if (fork()==0){
            dup2(fds[1], 1);        /* перенаправление вывода */
            close(fds[0]);
        execlp("who", "who", 0);
        }
```

```
/* Процесс-родитель закрывает оба конца конвейера и ожидает
завершения обоих
        дочерних процессов */
        else{
            close(fds[0]);
            close(fds[1]);
            wait(0);
            wait(0);
            }
}
```

При замещении дочерних процессов другими программами (посредством exec() вызова)
дескрипторы входа и выхода конвейера остаются обычно нетронутыми, и конвейер продолжает служить для передачи данных от одного процесса другому, посредством обычных системных вызовов *write()* и *read()*.

Конвейер представляется, как разновидность файла,
в котором можно сохранять ограниченный объем данных,
причем доступ к этим данным организован в манере
дисциплины обслуживания очереди типа FIFO (first-in first-out).
В большинстве систем конвейеры ограничены 10-ю логическими блоками по 512 байт.

Заголовочный файл <limits.h> содержит
предопределенную константу PIPE_BUF ,
задающую предельный размер буфера для каждой конкретной реализации. Система синхронизирует обращения процессов к конвейеру.
При попытке записи в заполненный канал, процесс будет автоматически заблокирован до тех пор, пока не освободится место для получения данных. Попытка чтения из пустого конвейера также приведет к блокировке, пока в конвейере не появятся данные.

Недостатками приведенного механизма передачи данных между процессами являются:
- невозможность аутентификации процесса
на другой стороне конвейера, то есть, процесс, читающий из *pipe*, не может знать, кто туда пишет;
- неудобно также и то, что конвейеры должны создаваться предварительно, еще до момента запуска процессов;
- обмен данными возможен лишь между родственными процессами, имеющими общего предка, передавшего потомкам файловые дескрипторы открытого конвейера;
- конвейеры не позволяют обмен данными по сети,
т.е. оба процесса должны выполняться на одном и том же компьютере.

Частично эти ограничения снимаются применением другой разновидности конвейеров, так называемых, *named pipes*.

Существует и другой путь создания обычных конвейеров для исполнения команд *shell*. Он основан на системных вызовах *popen()* и *pclose()*.

При вызове *popen()* выполняется последовательность сразу нескольких действий:

автоматически генерируется дочерний процесс, который, в свою очередь, запускает (*exec()*-ом) на исполнение команду *shell*, которая указана в качестве первого параметра *popen()*. Второй параметр вызова указывает, как интерпретировать дескриптор файла, указатель на который возвращает *popen()* в случае успешного завершения организации конвейера.

При указании “*w*” родительский процесс может писать данные в стандартный ввод команды *shell*, что дает возможность процессу потомку, выполняющему эту команду shell, читать эти данные (как стандартный ввод).

При указании “*r*” в качестве второго параметра *popen()*, наоборот, процесс родитель читает данные со стандартного вывода команды *shell*, запущенной дочерним процессом.

Программа *cmdpipe* иллюстрирует данный способ создания конвейера.

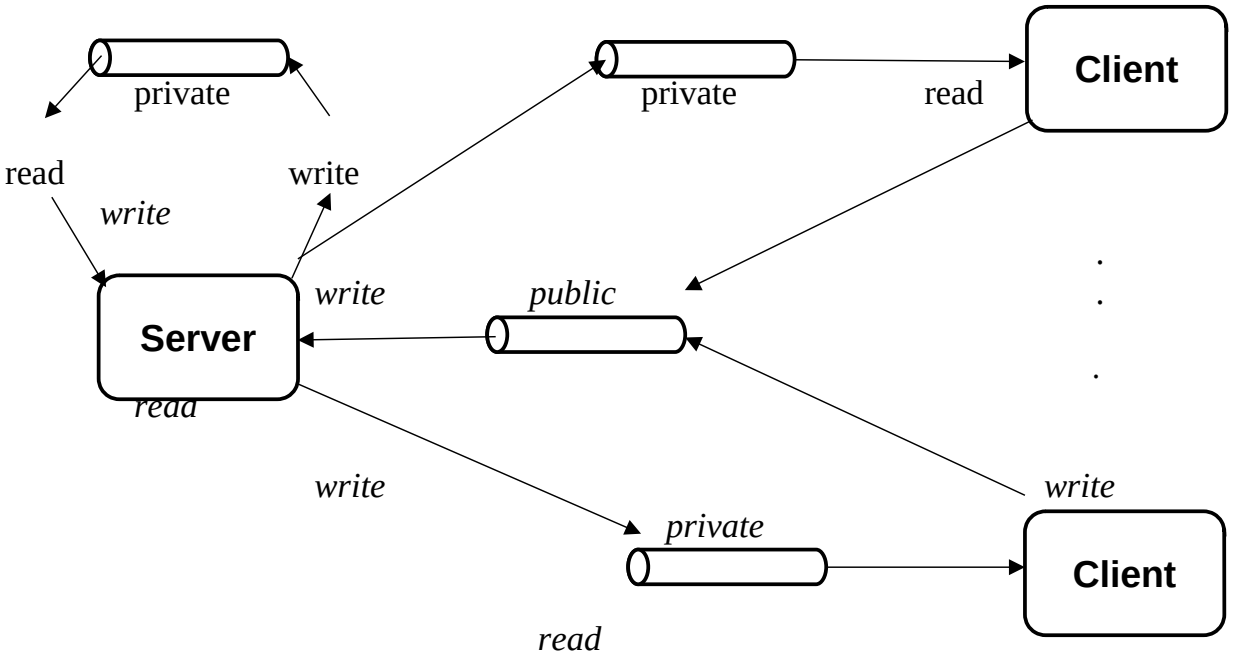
```
/* Программа cmdpipe.cpp */

/* Использование команд popen и pclose  */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<limits.h>
main()
{
FILE   *fin, *fout;
char   buffer[PIPE_BUF};
int     n;
   if(argc<3){
        fprintf(stderr, "Usage %s cmd1 cmd2\n", argv[0]);
        exit(1);
        }
fin  = popen(argv[1], "r");
fout = popen(argv[2], "w");
while((n = read(fileno(fin), buffer, PIPE_BUF))>0)
   write(fileno(fout), buffer, n);
pclose(fin);
pclose(fout);
exit(0);
}
```

Named pipes

Другой тип конвейера, называемый в оригинале *named pipe*, требует для своей организации системного вызова *mknod()*.
Вообще, системный вызов *mknod()* создает специальные файлы блочных или символьных устройств (не только конвейеры).
Именованные конвейеры (*named pipes*) в смысле организации взаимодействия между процессами, близки к обычным конвейерам (*pipes*), однако обладают некоторыми преимуществами. Так, при создании именованного конвейера, запись о нем, с соответствующими правами доступа, попадает в список файлов каталога. Это делает возможным применение таких конвейеров для коммуникации между любыми процессами, а не только между родственными. Процесс должен быть только наделен соответствующими правами.

Именованный конвейер может создаваться, как из приложения, так и на уровне *shell*.
Первый параметр системного вызова *mknod()* передает указатель на полное имя создаваемого конвейера в каталоге.
Второй параметр задает его тип. Для создания очереди *FIFO это* тип *S_FIFO*.
Второй параметр может также еще содержать логическое слагаемое, задающее права доступа к создаваемому ресурсу.
В списке каталога информация об атрибутах этого ресурса, помимо прав доступа к нему, содержит еще на первой позиции маркер "*p*", указывающий на то, что это конвейер (*pipe*).
Третий параметр *mknod()* задает тип устройства (для типа *S_FIFO* это - 0).
В программах *pipe_server* и *pipe_client приводится* реализация и совместное использование существующих механизмов обмена *pipes* и *named pipes* для организации взаимодействия процессов работающих в клиент-серверной парадигме. Заголовочный файл *pipe_local* является общим для обеих программ.
Иллюстрация построенной в данном примере схемы межпроцессного взаимодействия.



Процесс сервер запускается на исполнение в фоновом (*background*) режиме (символ *&* после имени запускаемого процесса).

Клиентские процессы запускаются последовательно в обычном режиме (*foreground*). Сервер создает именованный конвейер *PUBLIC*, доступный и для любого клиентского процесса.

Каждый же из клиентских процессов генерирует собственный именованный конвейер со своим уникальным именем. Далее клиент в ответ на свой запрос получает с консоли от пользователя команду *shell*, которую необходимо выполнить.

Затем клиент пишет в общедоступный именованный конвейер *PUBLIC* то уникальное имя, которое он присвоил собственному конвейеру при его создании, и пишет затем команду *shell*, полученную от пользователя.

На другом конце *PUBLIC* конвейера сервер читает эти данные от клиента. Сервер выполняет полученную команду *shell* с помощью системного вызова *popen()* (запускающего дочерний процесс для исполнения этой команды). По создаваемому тем же вызовом *popen()* собственному конвейеру сервер получает результат исполнения команды *shell*.
После этого сервер пишет в собственный именованный конвейер клиента результат исполнения команды *shell*, полученной ранее от данного клиента. Клиент читает эти данные и выводит их на *стандартный вывод ошибок* (чтобы можно было бы отличать данные, пришедшие клиенту от сервера).

```
/* Программа pipe_local.h */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<limits.h>
#include<string.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#define PUBLIC "/tmp/PUBLIC"
#define B_SIZ (PIPE_BUF / 2)
struct message{
char  fifo_name[B_SIZ];
char  cmd_line[B_SIZ];
};
```

```
/* Программа  pipe_server.cpp */
#include<pipe_local.h>
void main(void)
{
int     n, done, dummyfifo, privatefifo, publicfifo;
static char buffer[PIPE_BUF];
FILE *fin;
struct message msg;
       /* Создание очереди типа public FIFO */
mknod(PUBLIC, S_IFIFO | 0666, 0);
        /* Открыть public FIFO на чтение и запись */
if ((publicfifo=open(PUBLIC, O_RDONLY))==-1) ||
    (dummyfifo=open(PUBLIC, O_WRONLY | O_NDELAY))==-1){
   perror(PUBLIC);
   exit(1);
   }
              /* Сообщение можно прочитать из public конвейера */
while(read(publicfifo, (char *) &msg, sizeof(msg))>0){
   n = done = 0; /* Очистка счетчиков | флагов */
   do{                        /* Попытка открытия private FIFO */
       if ((privatefifo=open(msg.fifo_name, O_WRONLY | O_DELAY))==-1
       sleep(3);                    /* Задержка по времени  */
       else{                        /* Открытие успешно */
          fin = popen(msg.cmd_line, "r");/* Исполнение shell cmd,
полученной от клиента */
          write(privatefifo, "\n", 1);    /* Подготовка очередного вывода */
          while((n=read(fileno(fin), buffer, PIPE_BUF))>0){
              write(privatefifo, buffer, n);/* Вывод в private FIFO к клиенту */
              memset(buffer, 0x0, PIPE_BUF);     /* Очистка буфера */
              }
              pclose(fin);
              close(privatefifo);
              done = 1;                  /* Запись произведена успешно */
          }
       }while(++n<5 && !done);

       if(!done)                          /* Указание на неудачный исход */
          write(fileno(stderr), "\nNOTE: SERVER ** NEVER **
                             accessed private FIFO\n", 48);
   }
}
```

```
/* Программа  pipe_client.cpp */
#include<pipe_local.h>
void main(void)
{
int     n, privatefifo, publicfifo;
static char buffer[PIPE_BUF];
struct message msg;
                /* Создание имени для очереди типа private FIFO */
sprintf(msg.fifo_name, "/tmp/fifo %d", getpid());
                /* Создание очереди private FIFO */
if (mknod(msg.fifo_name, S_IFIFO | 0666, 0)<0){
   perror(msg.fifo_name);
   exit(1);
   }
                /* Открытие очереди типа public FIFO на запись */
if ((publicfifo=open(PUBLIC, O_WRONLY))==-1){
   perror(PUBLIC);
   exit(2);
   }
while(1){    /* Зацикливание */
   write(fileno(stdout), "\ncmd>", 6);
   memset(msg.cmd_line, 0x0, B_SIZ);            /* Очистка */
   n = read(fileno(stdin), msg.cmd_line, B_SIZ);        /* Чтение с консоли
shell cmd */
        if(!strncmp("quit", msg.cmd_line, n-1))
        break;                                  /*Завершение? */
   write(publicfifo, (char *) &msg, sizeof(msg)); /* to PUBLIC */
        /* Открытие private FIFO для чтения вывода исполненной команды
(shell cmd) */
        if((privatefifo = open(msg.fifo_name, O_RDONLY))==-1){
           perror(msg.fifo_name);
           exit(3);
           }
        /* Чтение private FIFO и вывод на результата на
стандартный_вывод_ошибок */
   while((n=read(privatefifo, buffer, PIPE_BUF))>0){
        write(fileno(stderr), buffer, n);
        }
        close(privatefifo);
   }
   close(publicfifo);
   unlink(msg.fifo_name);
}
```